Как решить проблемы оркестрации сотен задач по обработке данных с помощью Apache Airflow?

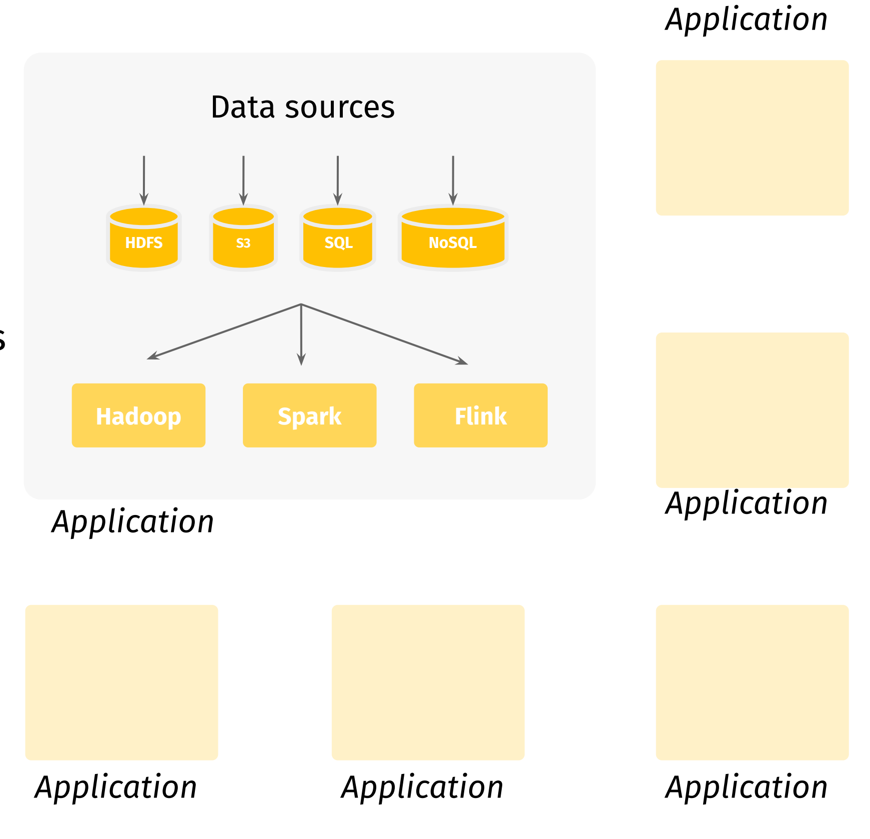
Владимир Баев





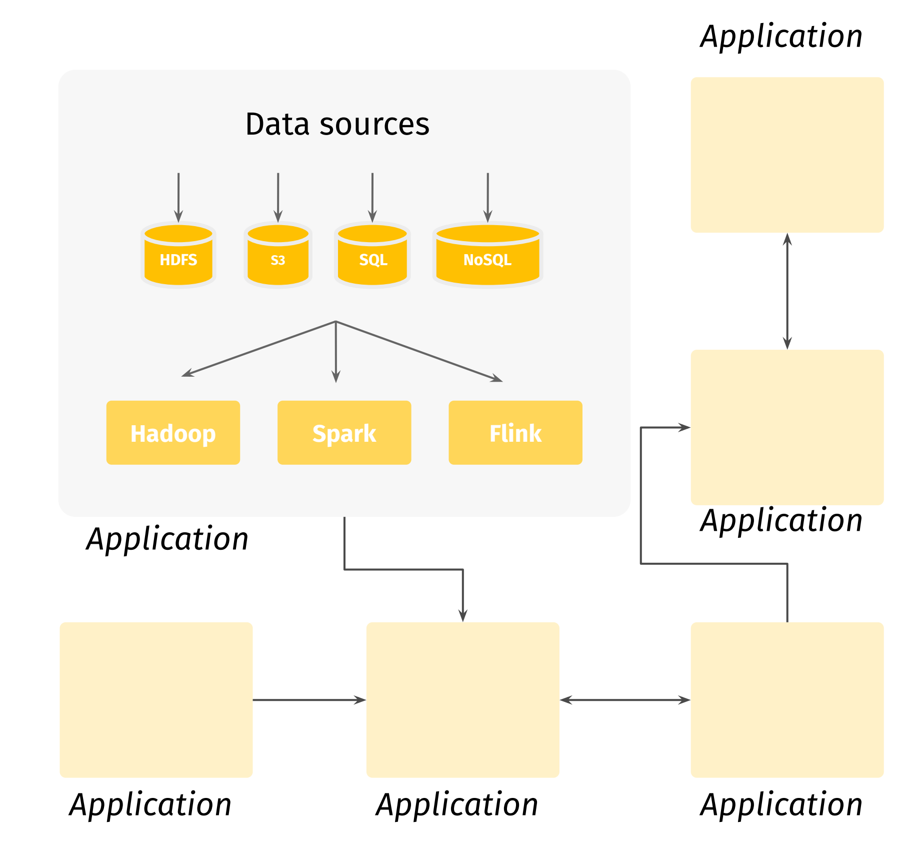
Big Data world: current state

- Problem
  - More data
  - More storage systems
  - More processing engines/tools
  - More application components
  - More dependencies between applications



Orchestration in Big Data

- Orchestration
  - Way of connecting components to provide appropriate scheduling and interaction
  - Example: sequence of Spark jobs, connected by inputs/outputs



About the author



Vladimir Baev

Program Manager
vbaev@griddynamics.com

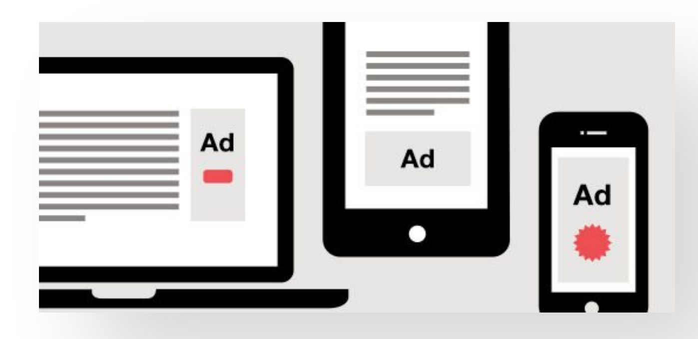
- 5+ years of experience in Big Data
- Working at Grid Dynamics
- Projects (adv analytics):
  - ETL Batch processing (Hadoop, Spark, Airflow)
  - ML platform automation (SparkML, H2O)

Agenda

1. Our project
2. Orchestration in Big Data world
3. Apache Airflow overview
4. Production use cases
5. Issues, caveats and tips
6. Extensions
7. Resources

Our project

- Digital advertising in the cloud: analytics, reporting, suspicious activity detection, ML models
  - 30 TBs of raw data daily
  - 15 billions of input events daily
  - Various SLAs from 10 minutes to few hours
  - Batch processing
  - Machine Learning, ETL applications
  - MapReduce, Hive, Spark

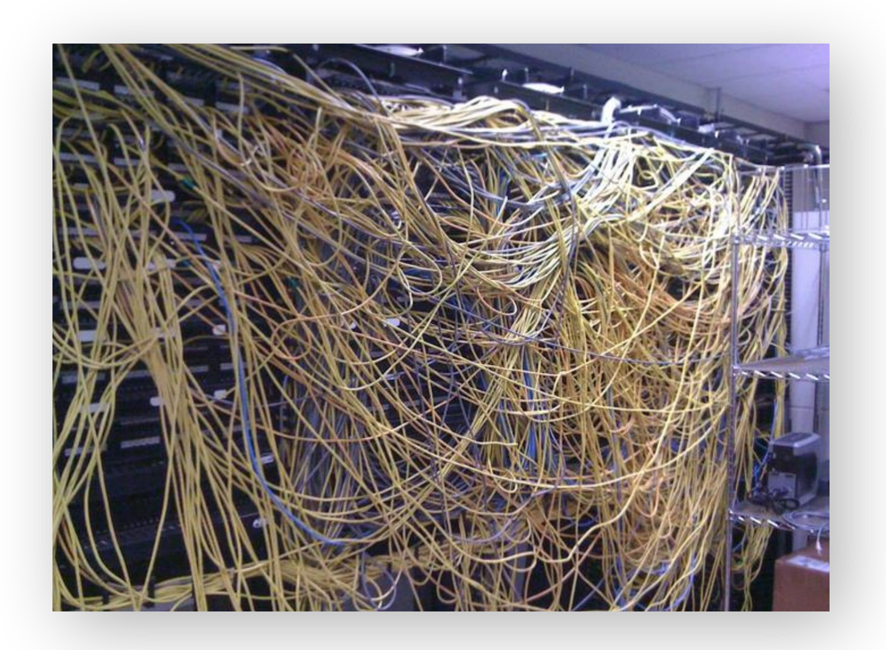


Orchestration in such constraints is a real challenge!

Orchestration in Big Data world

Orchestration in Big Data world: requirements

- Support of various sources
  - Web
  - Media
  - Transactions
- Reliable
- Distributed
- Fault-tolerant
- Scalable
- Reproducible
- Flexible, customizable
- Monitoring, restatement



Orchestration in Big Data world: major players















Orchestration in Big Data world: our case

Our case

60 applications in 5 projects
Codebase age: 1 month to 5 years

Existing solution

Mix of various orchestration tools (cron, luigi, oozie, in-house scheduler), spread across few clusters

Issues

Maintainability
Fault-tolerance
Monitoring, restatement

Desired goal

Unification
Reliable tooling
Ease of support for duty engineers
Transparent deployments and versioning

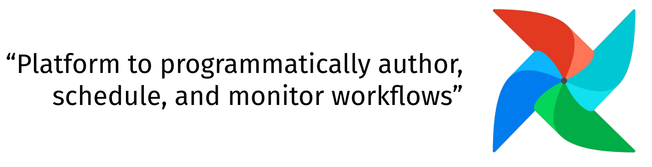
Solution

Apache Airflow?

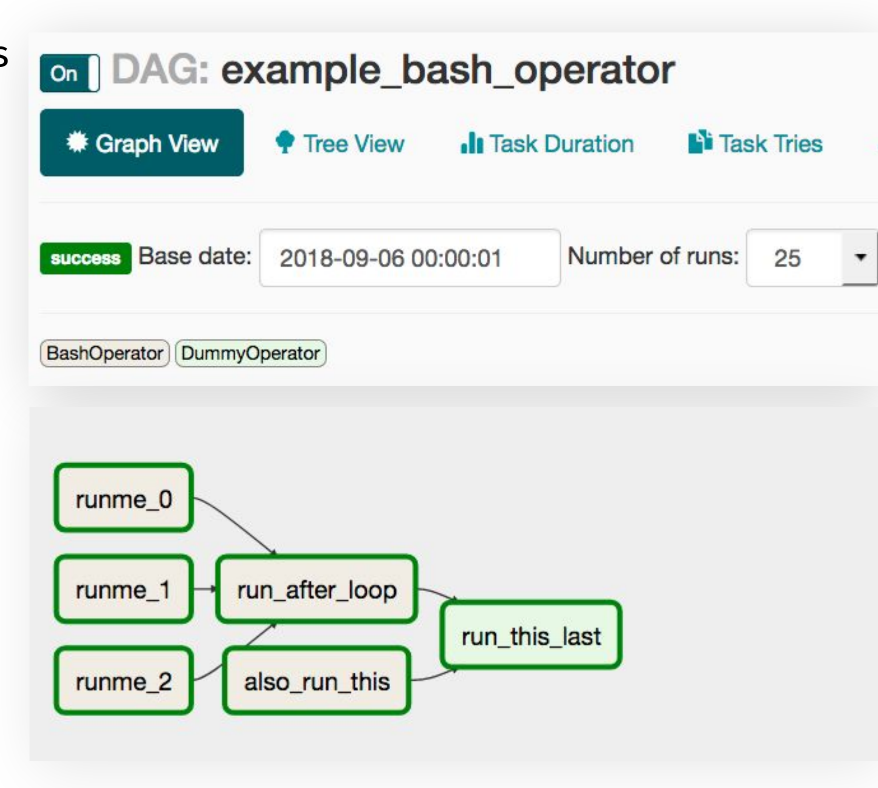
Let’s try!

Introduction
to Apache Airflow

Apache Airflow: Overview



- Vocabulary
  - DAGs (Directed acyclic graphs) = pipelines
  - Tasks, Task instances
  - Operators, Sensors
  - DAG run
  - master, worker nodes
- Features
  - Batch-oriented
  - Define pipelines (DAGs) as Python code
  - Dynamic DAGs support
  - Extensible
  - Parameterizing with Jinja templates
  - Scalable
  - Rich UI

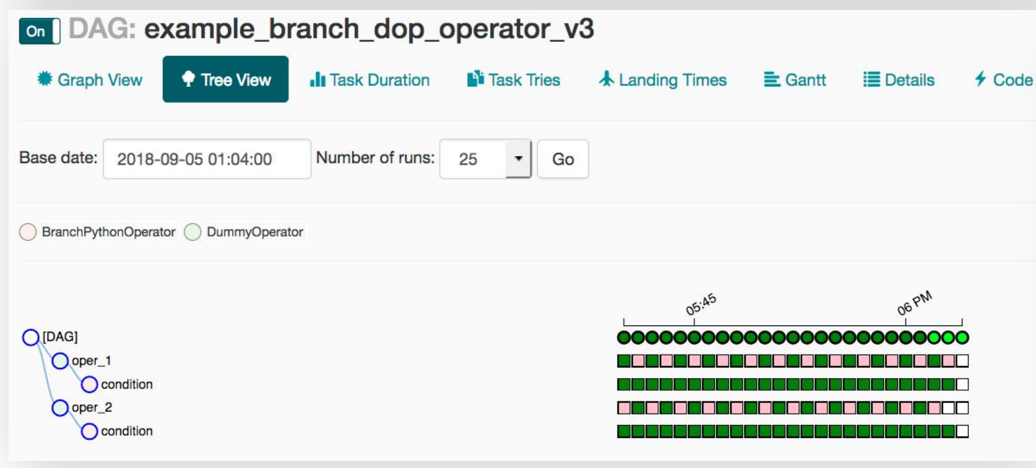


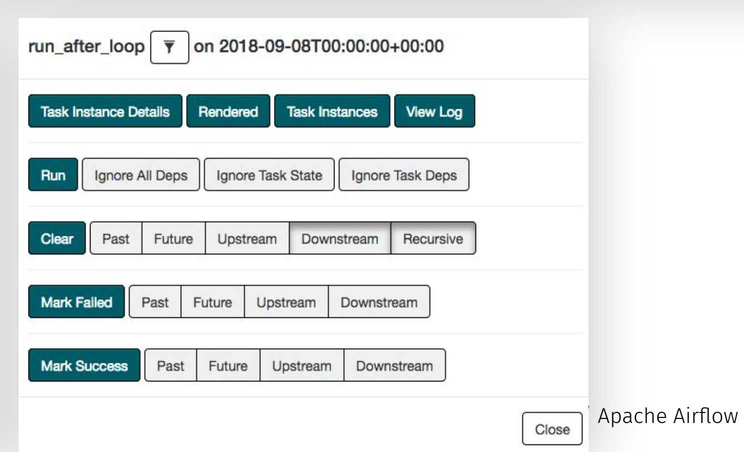
Apache Airflow: Overview

DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	On	example_bash_operator	0 0 * * *	airflow	6	2018-09-06 00:00	5	
	On	example_branch_dop_operator_v3	*/1 * * * *	airflow	3 1 1 5	2018-09-05 00:56	54 3	
	On	example_branch_operator	@daily	airflow	5	2018-09-06 00:00	2	
	On	example_xcom	@once	airflow	3	2018-09-05 00:00	1	
	On	latest_only	4:00:00	Airflow	2	2018-09-07 16:00	35	





Apache Airflow: Concepts

- Operators and Sensors
  - BashOperator
  - PythonOperator
  - PostgresOperator
  - SparkSubmitOperator
  - BaseBranchOperator, TriggerDagRunOperator, SubDagOperator
  - S3PrefixSensor
- Pools, Queues
- Hooks (HDFSHook, SlackHook), Connections, Variables, XComs, etc

Simple DAG

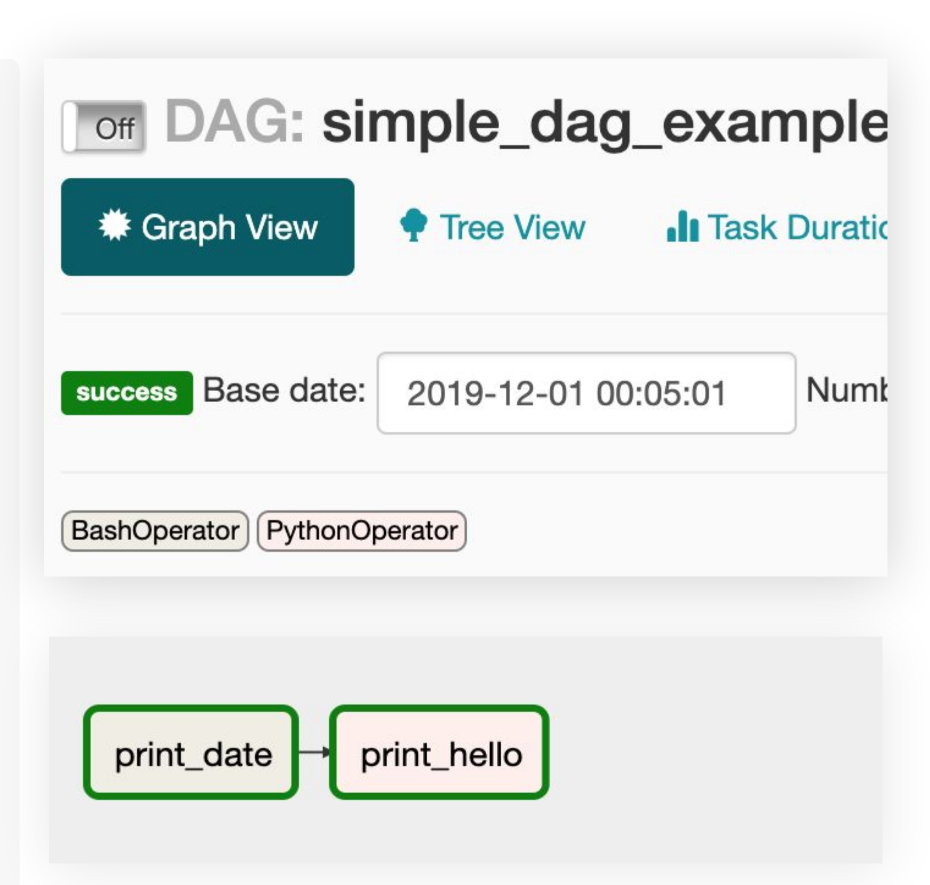
```python
dag = DAG(
    'simple_dag_example',
    default_args=default_args,
    description='A simple tutorial DAG',
    schedule_interval='*/5 * * * *',
)

t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag
)

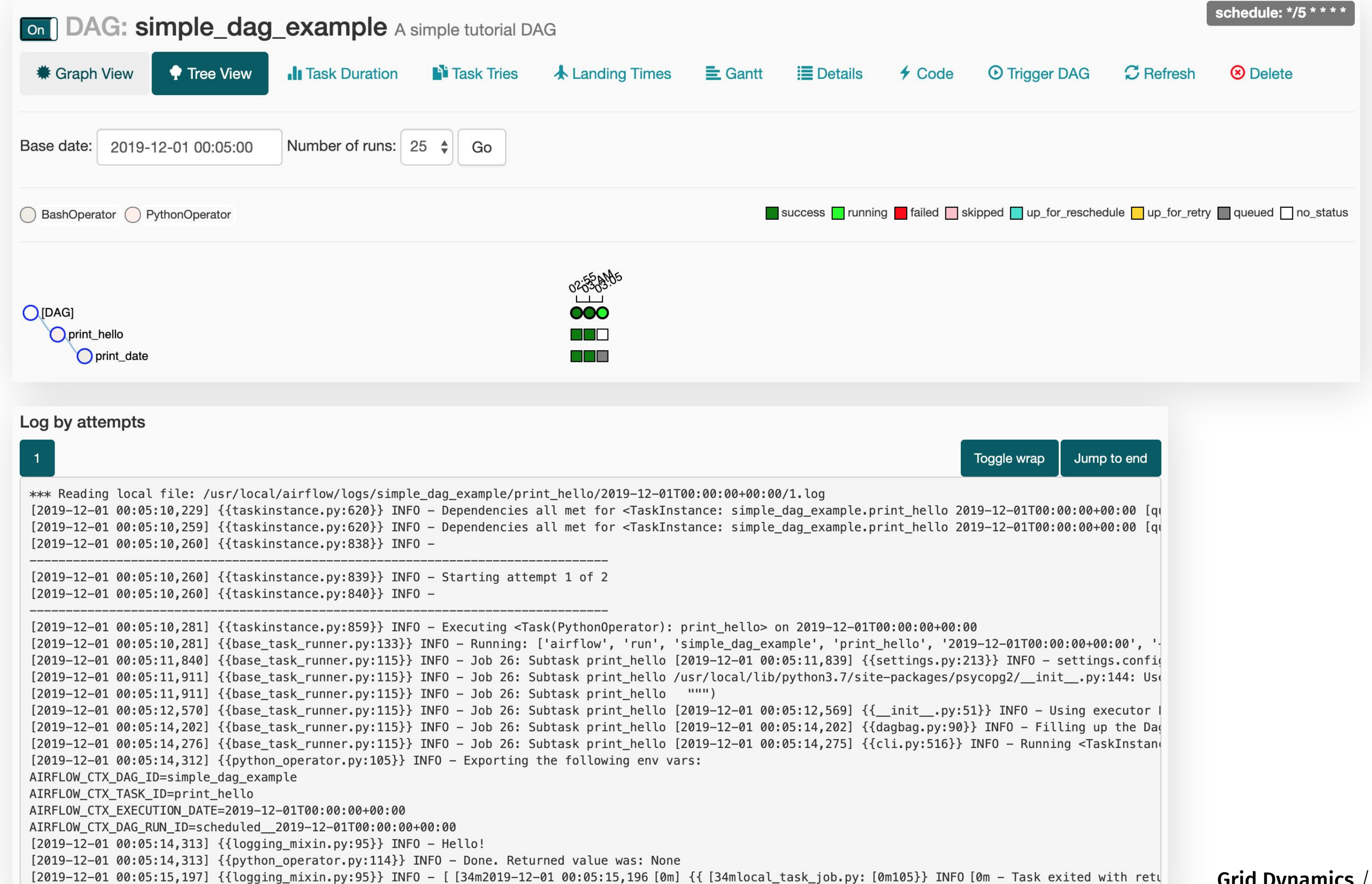
def print_hello():
    print('Hello!')

t2 = PythonOperator(
    task_id='print_hello',
    python_callable=print_hello,
    dag=dag
)

t1 >> t2
```



Simple DAG



schedule: */5 * * * *
On
DAG: simple_dag_example A simple tutorial DAG
Graph View
Tree View
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
Trigger DAG
Refresh
Delete
Base date: 2019-12-01 00:05:00
Number of runs: 25
Go
BashOperator
PythonOperator
success
running
failed
skipped
up_for_reschedule
up_for_retry
queued
no_status
[DAG]
print_hello
print_date
Log by attempts
1
Toggle wrap
Jump to end
*** Reading local file: /usr/local/airflow/logs/simple_dag_example/print_hello/2019-12-01T00:00:00+00:00/1.log
[2019-12-01 00:05:10,229] {{taskinstance.py:620}} INFO - Dependencies all met for <TaskInstance: simple_dag_example.print_hello 2019-12-01T00:00:00+00:00 [q
[2019-12-01 00:05:10,259] {{taskinstance.py:620}} INFO - Dependencies all met for <TaskInstance: simple_dag_example.print_hello 2019-12-01T00:00:00+00:00 [q
[2019-12-01 00:05:10,260] {{taskinstance.py:838}} INFO -

[2019-12-01 00:05:10,260] {{taskinstance.py:839}} INFO - Starting attempt 1 of 2
[2019-12-01 00:05:10,260] {{taskinstance.py:840}} INFO -

[2019-12-01 00:05:10,281] {{taskinstance.py:859}} INFO - Executing <Task(PythonOperator): print_hello> on 2019-12-01T00:00:00+00:00
[2019-12-01 00:05:10,281] {{base_task_runner.py:133}} INFO - Running: ['airflow', 'run', 'simple_dag_example', 'print_hello', '2019-12-01T00:00:00+00:00', '
[2019-12-01 00:05:11,840] {{base_task_runner.py:115}} INFO - Job 26: Subtask print_hello [2019-12-01 00:05:11,839] {{settings.py:213}} INFO - settings.confi
[2019-12-01 00:05:11,911] {{base_task_runner.py:115}} INFO - Job 26: Subtask print_hello /usr/local/lib/python3.7/site-packages/psycopg2/__init__.py:144: Us
[2019-12-01 00:05:11,911] {{base_task_runner.py:115}} INFO - Job 26: Subtask print_hello """)
[2019-12-01 00:05:12,570] {{base_task_runner.py:115}} INFO - Job 26: Subtask print_hello [2019-12-01 00:05:12,569] {{__init__.py:51}} INFO - Using executor
[2019-12-01 00:05:14,202] {{base_task_runner.py:115}} INFO - Job 26: Subtask print_hello [2019-12-01 00:05:14,202] {{dagbag.py:90}} INFO - Filling up the Da
[2019-12-01 00:05:14,276] {{base_task_runner.py:115}} INFO - Job 26: Subtask print_hello [2019-12-01 00:05:14,275] {{cli.py:516}} INFO - Running <TaskInstan
[2019-12-01 00:05:14,312] {{python_operator.py:105}} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_ID=simple_dag_example
AIRFLOW_CTX_TASK_ID=print_hello
AIRFLOW_CTX_EXECUTION_DATE=2019-12-01T00:00:00+00:00
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2019-12-01T00:00:00+00:00
[2019-12-01 00:05:14,313] {{logging_mixin.py:95}} INFO - Hello!
[2019-12-01 00:05:14,313] {{python_operator.py:114}} INFO - Done. Returned value was: None
[2019-12-01 00:05:15,197] {{logging_mixin.py:95}} INFO - [[34m2019-12-01 00:05:15,196 [0m] {{ [34mlocal_task_job.py: [0m105}} INFO [0m - Task exited with retu

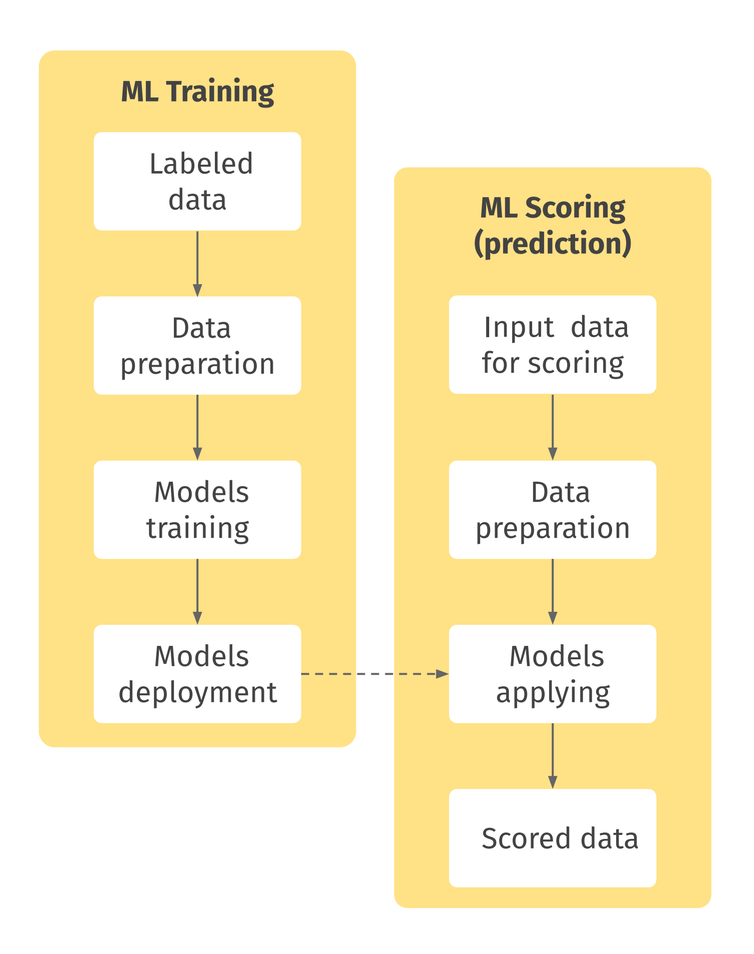
Apache Airflow in Production System

Airflow in production: checklist

1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Business logic
Example of production DAGs

- ML models training workflow
  - Wait for new raw training data
  - Process training data, add to training dataset
  - Launch ML model training
  - Release new model
- ML scoring workflow (batch predictions)
  - Wait for input batches to score (S3)
  - Trigger scoring dag for each input
    - Spark scoring with appropriate models
    - Metrics export
  - Mark batches as done

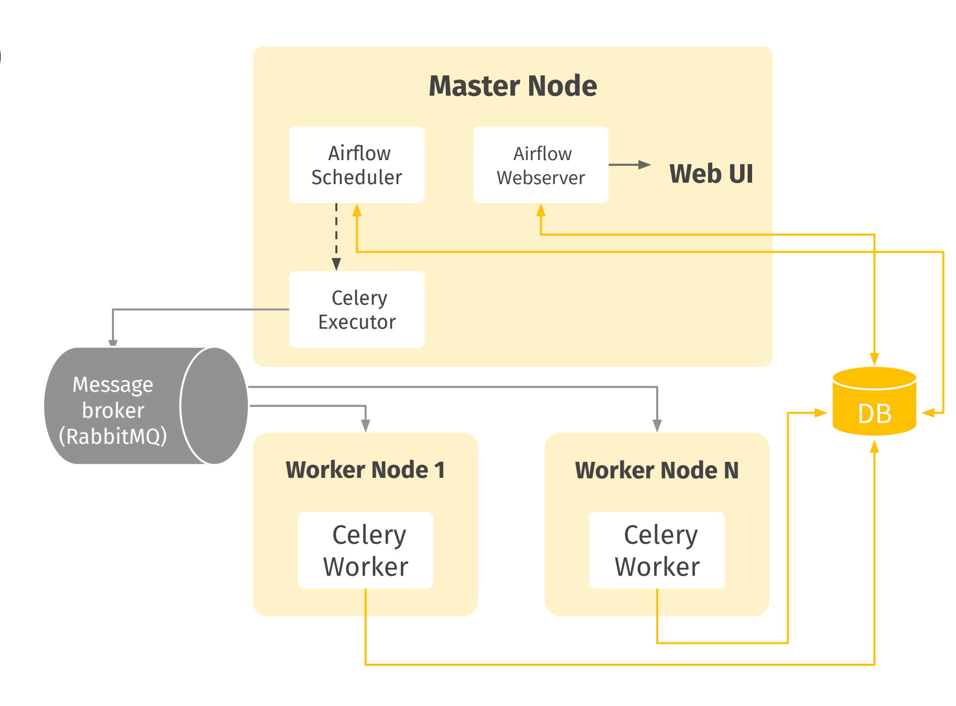


Airflow in production: checklist

1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Example of Airflow setup

- 3 Airflow masters (dev/staging/prod environments)
- Each master have:
  - Airflow scheduler, webserver
  - 20 Airflow workers (CeleryExecutors)
  - 180 DAGs
  - 5 projects
- Python virtualenvs
  - Airflow runs scheduler and executes tasks under virtualenvs
    - scheduler env
    - workers env
  - PythonVirtualenvOperator



Infrastructure

Challenges of Airflow migrations

Migration from Airflow 1.8 to 1.10.10

- updates of custom Operators, pipelines
- DB migration

Migration from Python 2.7 to 3.6

- duplicate virtualenvs, airflow masters, workers
- run both python2 and 3 processes on workers

Future: Airflow 2.0 migration

- lose backward compatibility with 1.x versions

Infrastructure

Airflow Integration

- Executors
    - SequentialExecutor
    - LocalExecutor
    - CeleryExecutor
    - Kubernetes Executor
- Cloud integrations (Operators, Hooks)
    - Azure
    - AWS
    - GCP
- Command Line Interface
- REST API (experimental)
- Managed Airflow as a service
    - Astronomer
    - GCP, AWS

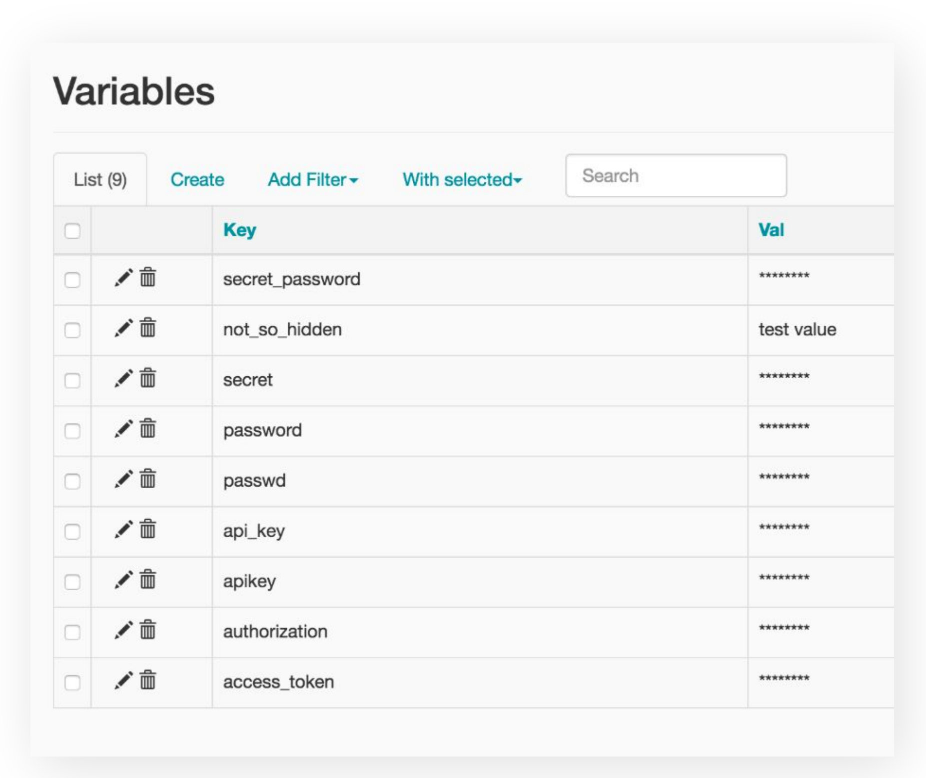
Airflow in production: checklist

1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Flexibility

Configuration: code vs variables

- Pipeline's configuration: redeployment required
- Airflow variables

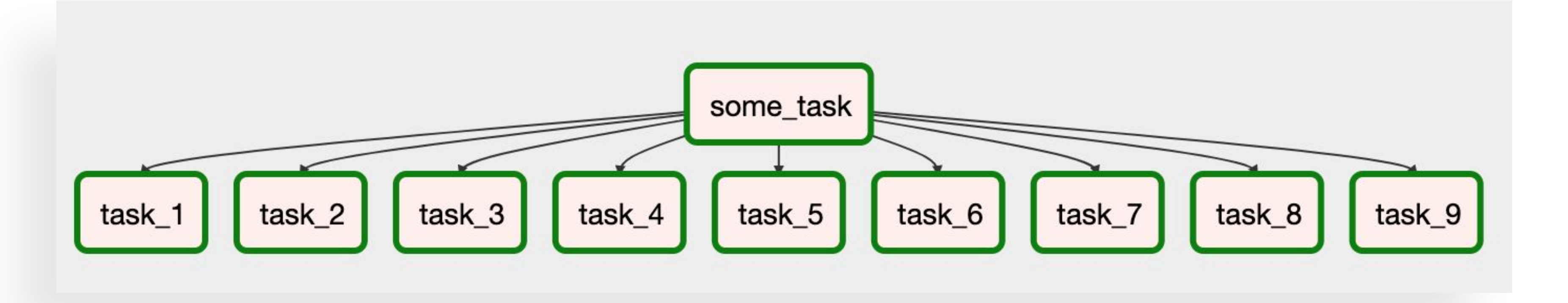


Flexibility

Dynamic tasks

```python
t_0 = PythonOperator(task_id='some_task', python_callable=foo, dag=dag)

for idx in range(1, 10):
    t_i = PythonOperator(task_id=f'task_{idx}', python_callable=foo, dag=dag)
    t_i.set_upstream(t_0)
```



Flexibility

Dynamic DAGs

```python
def create_dag(dag_id):
    dag = DAG(dag_id, default_args=default_args, schedule_interval=None)
    task = PythonOperator(task_id='some_task', python_callable=foo, dag=dag)
    return dag

for idx in range(1, 10):
    dag_id = f'dynamic_dag_{idx}'
    globals()[dag_id] = create_dag(dag_id)
```

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs
	On	dynamic_dag_1	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_2	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_3	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_4	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_5	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_6	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_7	None	airflow	1	2019-11-30 23:44	1
	On	dynamic_dag_8	None	airflow	1	2019-11-30 23:45	1
	On	dynamic_dag_9	None	airflow	1	2019-11-30 23:45	1

Airflow in production: checklist

1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Testing

- Local development
  - Airflow Docker image
  - Airflow setup
- Unit testing
  - Test that DAG “compiles”
  - Test structure of DAG’s tasks
  - Test Operators and tasks
- End-to-end testing with REST API
- Separate DAGs for integration testing

```python
def test_dag(dag_id, task_id_1, task_id_2):
    dag_bag = DagBag(dag_folder=dag_folder, executor='LocalExecutor')
    assert len(dag_bag.import_errors) == 0
    assert dag_id in dag_bag.dags
    dag = dag_bag.get_dag(dag_id)
    task_1 = dag.get_task(task_id_1)
    assert task_1
    downstream_tasks = [t.task_id for t in task_1.downstream_list]
    assert downstream_tasks == [task_id_2]

    task = pipeline.dag.task_dict['move_file']
    task.templates_dict = dict(src_path=src_path, dst_path=dst_path)
    ti = TaskInstance(task=task, execution_date=datetime.now())
    task.execute(ti.get_template_context())
    assert os.path.exists(dst_path)
```

Airflow in production: checklist

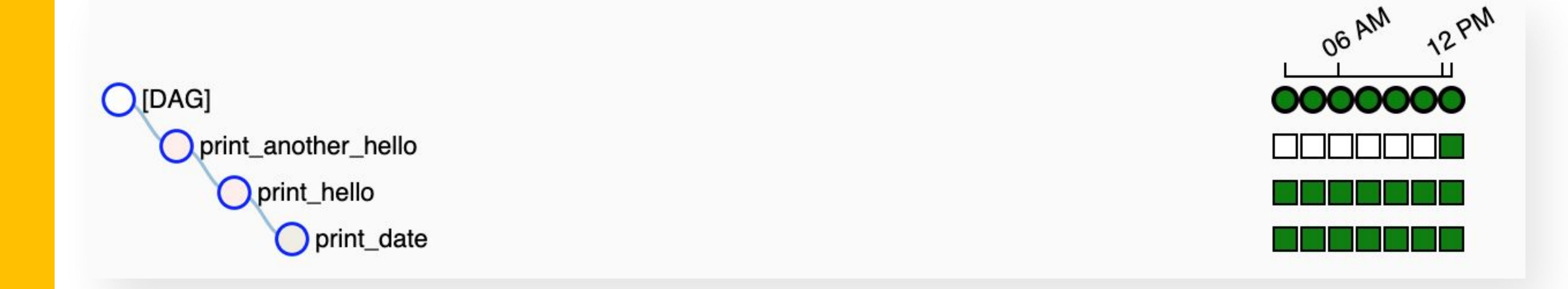
1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Deployment

DAG versioning

- New DAG after update
  - create DAG with new name
  - manage actual version
- Modifying existing DAG
  - new tasks
  - updates of running DAGs

Off	simple_dag_example_v2.0	*/5 * * * *	airflow	2
On	simple_dag_example_v2.1	*/5 * * * *	airflow	2



DAG management

Deployment

update artifact (pipeline and dependencies) on master and workers hosts

Issues

- consistency between nodes
- updates of running DAGs

DAG management via DB

- pause DAG scheduling
- wait for completion of dag_runs
- deploy new version
- resume DAG scheduling

Airflow in production: checklist

1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Optimisation

Execution tuning by airflow.cfg

- parallelism: number of task instances per worker
- concurrency: number of scheduled task instances for DAG
- max_active_runs: number of running DagRuns for DAG
- celery configurations
- scheduler configurations
  - max_threads
  - scheduler_heartbeat_sec
- .airflowignore



Airflow in production: checklist

1. Business logic
2. Infrastructure
3. Flexibility
4. Testing
5. Deployment
6. Optimisation
7. Monitoring

Monitoring

- Airflow sends emails on DAGs failures or retries
- Tasks monitoring
  - sla - time by which the job is expected to finish
  - execution_timeout - max time allowed for the execution of this task instance
  - retries, retry_delay - multiple attempts for task execution
- DAGs monitoring:
  - dagrun_timeout - max time allowed for the execution of dag_run
  - sla_miss_callback - a function to call when reporting SLA timeouts
- SLAs view

But what if we need even more?

Apache Airflow Extensions

Extensions mechanism

- Custom operators, sensors, hooks
- Custom views, UI elements
- Plugins for integration and auto-importing



YAML Config

Issues

- Hardcoded values
- Code duplication
- Overcomplicated pipeline’s code

YAML Config

Issues

- Hardcoded values
- Code duplication
- Overcomplicated pipeline's code

Solution

- Integrate jinja airflow templates into YAML parser
- Reuse base config and override for dev/staging/prod environments

```yaml
# base_config.yaml
dag:
  start_date: !datetime 2019-01-01 00:00
  email: !airflow_variable mailing_list
  execution_date: !execution_date
  output: !from_xcom
    key: output
  env: !override_required


# dev_config.yaml
include: !include_parsed base_config.yaml

env: !override
  key: include.dag.env
  value: dev
```

Autodocs

Issues

Outdated documentation for duty engineers

Lack of standardization

Autodocs

Issues

Outdated documentation for duty engineers

Lack of standardization

Solution

Integration of Sphinx documentation into Airflow pipelines
- DAG name, scheduling interval
- input/output paths
- Visualisation of tasks

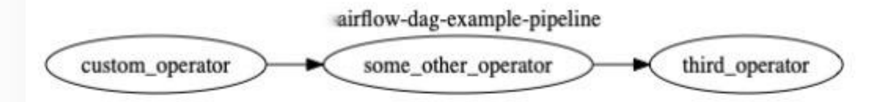
Input/Output Storage

Environment	Storage Type	Storage URL	Data TTL, Days	Data Volume, GB
prod	output	/prod/out	1	10

Pipeline's Summary

This pipeline moves data from the local FS to the database.

Airflow DAG



Pipeline autodocs

Main pipeline code

class **pipeline.CustomOperator**(**args*, ***kwargs*)[source]
This is first dummy operator's custom operator class

class **pipeline.SomeOtherOperator**(**args*, ***kwargs*)[source]
Custom operator for some_other_operator

Monitoring

Issue

State of DAGs in case of Airflow outage

Goal

Provide reliable external monitoring system

Monitoring

Issue

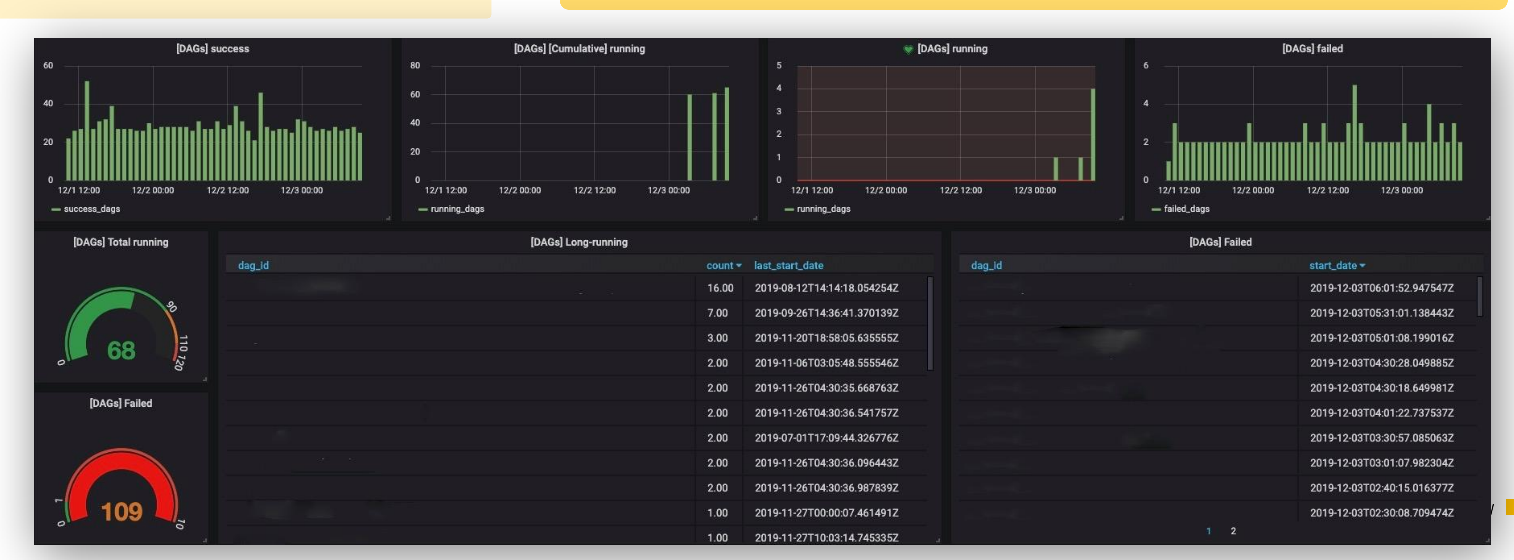
State of DAGs in case of Airflow outage

Goal

Provide reliable external monitoring system

Solution

- Dump logs from scheduler, tasks to ElasticSearch
- Airflow database
- Grafana dashboards
- Zabbix hooks to handle tasks completion



Conclusion

Conclusion

- Apache Airflow successfully works as an orchestration tool and initiated process of tools unification in our project
  - scalable, fault-tolerant
  - flexible and customizable
  - mature enough to handle production workloads
  - have a good community
  - In our project it significantly reduced number of production incidents
  - Grid Dynamics actively contributes to Airflow
- Next steps
  - Migration from on-prem to AWS
  - Airflow on Kubernetes
  - Common library for AWS pipelines
  - Self-service for infrastructure provisioning and scaling
  - Migration to Airflow 2.0

Resources

- https://airflow.apache.org/
- https://github.com/apache/airflow
- https://stackoverflow.com/questions/tagged/airflow/
- Common Pitfalls: https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=62694614
- Official Slack workspace: https://apache-airflow.slack.com/
- Russian Telegram community (1200 members): https://t.me/ruairflow



Q&A

Thank you!

